Lexical Analysis, I

Comp 412

Chapter 2 in EaC2e
Adjusted Calendar

Lab 1, Adjusted Schedule

Code Check 1  Monday, September 11, 2017
Code Check 2  Monday, September 18, 2017
Due Date for Code  Monday, September 25, 2017
Last Day for Code  Monday, October 2, 2017

Midterm Exam  Wednesday, October 18 @ 7PM (unchanged)

Lab 3, Adjusted Schedule

Lab 3 Available  Friday, October 20, 2017
Due Date for Code  Wednesday, November 15, 2017
Last Day for Code  Wednesday, November 22, 2017
The Front End

Scanner looks at every character
- Converts stream of chars to stream of classified words:
  - \langle\text{category}, \text{lexeme}\rangle
  - Sometimes call this pair a “token”
- Efficiency & scalability matter

Parser looks at every token
- Determines if the stream of tokens forms a sentence in the source language
- Fits tokens to some syntactic model, or grammar, for the source language
The Front End

Why separate scanning & parsing?
• Primary rationale is efficiency
• Scanner identifies & classifies words by their spelling
  – Abstracts spelling into category
• Parser constructs derivations
• Parsing is harder than scanning

Modern view (*less widely held*)
• Scanner-less parsers are gaining popularity, because they eliminate one more set of tools
  – Maybe we can afford the overhead
  – A little more involved (SGLR parsers)
How do we automate the construction of scanners & parsers?

**Scanner**
- Specify syntax with regular expressions (REs)
- Construct finite-automaton & scanner from the RE

**Parser**
- Specify syntax with context-free grammars (CFGs)
- Construct push-down automaton & parser from the CFG
How Does Class Relate to Regex Libraries?

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. ...

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

From Python 2.7.10 documentation, emphasis added

• You will learn how to “compile” REs to a DFA & implement a DFA
  – Execution cost is guaranteed $O(1)$ per input character, independent of the expression
• You will have deeper understanding of their power & their use
In Lecture 2, we saw some ambiguity in defining “positive integer”
• Is 001 a positive integer? What about 00?
• The automata are precise specifications, but the words are not

We need a better notation for specifying microsyntax than these transition diagrams.
Regular Expressions

We need a better notation for specifying microsyntax

Regular Expressions over an Alphabet \( \Sigma \)

- If \( x \in \Sigma \), then \( x \) is an RE denoting the set \( \{ x \} \) or the language \( L = \{ x \} \)
- If \( x \) and \( y \) are REs then
  - \( xy \) is an RE denoting \( L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \} \)
  - \( x | y \) is an RE denoting \( L(x) \cup L(y) \)
  - \( x^* \) is an RE denoting \( L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k \) (Kleene Closure)
    \( \rightarrow \) Set of all strings that are zero or more concatenations of \( x \)
  - \( x^+ \) is an RE denoting \( L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k \) (Positive Closure)
    \( \rightarrow \) Set of all strings that are one or more concatenations of \( x \) (or \( xx^* \))
- \( \epsilon \) is an RE denoting the empty set

Many RE-based systems support additional notation and operators. Those added features build on alternation, concatenation, and closure — plus, perhaps logical complement or negation. Complement is easy and efficient, if we think of the underlying DFA. (We will revisit this issue.)
Regular Expressions

How do these operators help?

Regular Expressions over an Alphabet $\Sigma$

- If $x$ is in $\Sigma$, then $x$ is an RE denoting the set $\{ x \}$ or the language $L = \{ x \}$
  \[ \text{The spelling of any letter in the alphabet is an RE} \]
- If $x$ and $y$ are REs then
  - $xy$ is an RE denoting $L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \}$
    \[ \text{If we concatenate letters, the result is an RE, so we can spell words} \]
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
    \[ \text{Any finite list of words can be written as an RE, } (w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n) \]
  - $x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$
  - $x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$
    \[ \text{We can use closure to write finite descriptions of infinite, but countable, sets} \]
- $\epsilon$ is an RE denoting the empty set
  \[ \text{\epsilon is sometimes useful for writing more concise REs} \]
Regular Expressions

Let the notation \([a...z]\) be shorthand for the \(\text{RE}\)

\[
(a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z)
\]

Examples

- **Tasteless positive integer**
  
  \[0...9] [0...9]^*

  or

  \[0...9]^*

- **Tasteful positive integer**
  
  \(0 \mid [1...9] [0...9]^*\)

- **Identifier (Algol-like lang)**
  
  \(([a...z] | [A...Z]) (\([a...z] | [A...Z] | [0...9]\))^*\)

- **Decimal number**
  
  \(0 \mid [1...9] [0...9]^* \cdot [0...9]^*\)

- **Real number**
  
  \(((0 \mid [1...9] [0...9]^* ) \mid (0 \mid [1...9] [0...9]^* \cdot [0...9]^* )) \cdot [0...9]^*\)
What Is The Point?

Why do we care about regular expressions in the context of a compiler?

• We use REs to specify the mapping of words to parts of speech
  – An identifier is \(( [a..z] \mid [A..Z] ) ( [a..z] \mid [A..Z] \mid [0..9] )^*\)
  – Keywords are specified by their spellings, e.g., `if`, `then`, `else`

• We use tools derived from automata theory to construct scanners directly from the REs
  – Automatic construction reduces the time & cost of scanner construction
  – Derivation from a formal notation eliminates implementation errors
  – Resulting scanners are both **efficient** \(O(n)\) and **fast** (low constant overhead)

• RE-derived scanners are widely used
  – Compilers, text editors
  – Input checking in many contexts
  – Software to filter or block **URLs**

We typically add some special characters, e.g., `_, #, $, @`
A Digression on Time

In COMP 412, we will talk about a lot of “times”

• Design time, implementation time, compile time, run time, ...
• In practice, the issue of when something happens is one that causes a great deal of confusion among students of compiler construction

  ➝ Design time and build time happen long before compiler runs
  ➝ Costs incurred at design or implementation time do not increase compile time

  ➝ Compile time happens every time the user invokes the compiler
  ➝ Users are, appropriately, sensitive to compile time
  ➝ Costs incurred at compile time do not increase run time

  ➝ Run-time costs affect actual application performance
  ➝ One critical goal for compilation is to keep run time to a minimum, which means reducing the overhead introduced by translation

As we look at strategies for generating scanners & parsers, keep in mind that generation costs are incurred at implementation time
Automatic Scanner Construction

1. We write REs at design time
2. Tools generate the scanner at build time
   specifications written as regular expressions
3. When the compiler runs, it uses the generated scanner to convert source code into a stream of tokens.

Scanner

generate Scanner Generator

• Simplify the construction of robust, efficient scanners
• Develop techniques that have widespread applicability
• Understand the underlying theory & practice
Automatic Scanner Construction

Scanner Generator

- May encode its knowledge in tables that drive a “skeleton scanner”
  - Skeleton scanner interprets the tables to simulate the DFA
  [See § 2.5.1](#)
- Every scanner uses the same skeleton
- Scanner generator builds the DFA from the RE, & converts it to a table

source code → Skeleton Scanner → Tables → <word, category> pairs

specifications (as REs) → Scanner Generator

Knowledge encoded in tables to drive skeleton
Automatic Scanner Construction

Scanner Generator

• May encode its knowledge of the recognizer directly into code
  – Transitions are compiled into conditional logic
  
  See § 2.5.2

• Produces a scanner that has very low overhead per character

• Scanner generator builds the DFA from the RE, & emits code for it
Example from Lecture 2

Recognizer for an ILOC register name (allow redundant zeros)

Rules for DFA Operation
- Start in state $s_0$ & make transitions on each input character
- DFA accepts a word $x$ if and only if $x$ leaves the DFA in a final state $s_i$
- If the DFA encounters a character with no specified transition, it moves to $s_e$ & stays in that state
- $r17$ takes it through $s_0, s_1, s_2, s_2$ and it accepts
- $r$ takes it through $s_0, s_1$ and it fails
- $ra$ takes it through $s_0, s_1, s_e$ and it fails

We will use the RE for a register name as a continuing example.
Example

To be useful, the DFA must be executable

char ← next character
state ← s₀
while (char ≠ EOF) {
  state ← δ[state,char]
  char ← next character
}
if (state is a final state) then report success else report failure

O(1) per character

Skeleton Scanner

Transition Table (δ)

For each character, the skeleton scanner does a table lookup and reads the next character — both of which should be O(1) operations

This skeleton scanner is simplified. See Figure 2.14 in § 2.5.1 of EaC2e.
Example

To capture and classify the lexeme, we add a little work to each state

\[
\begin{align*}
\text{char} & \leftarrow \text{next character} \\
\text{state} & \leftarrow s_0 \\
\text{lexeme} & \leftarrow \text{null string} \\
\text{while} (\text{char} \neq \text{EOF}) & \{
\quad \text{lexeme} \leftarrow \text{lexeme} \mid \mid \text{char} \\
\quad \text{state} \leftarrow \delta[\text{state},\text{char}] \\
\quad \text{char} \leftarrow \text{next character}
\}\n\end{align*}
\]

If (state is a final state) then {
\[
\begin{align*}
\text{category} & \leftarrow f(\text{state}) \\
\text{return} & <\text{lexeme},\text{category}> \\
\end{align*}
\]
} else report failure

\[
\begin{array}{|c|c|c|c|}
\hline
\delta & r & 0,1,2,3,4,5,6,7,8,9 & \text{Any Other} \\
\hline
s_0 & s_1 & s_e & s_e \\
\hline
s_1 & s_e & s_2 & s_e \\
\hline
s_2 & s_e & s_2 & s_e \\
\hline
s_e & s_e & s_e & s_e \\
\hline
\end{array}
\]

Transition Table (\(\delta\))
Example

To capture the register number, we would need state-specific actions

```
char ← next character
state ← s₀
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← next character
    if (state = s₁)
        n ← 0
    else if (state = s₂)
        n ← n * 10 + char – ‘0’
}

If (state is a final state) then {
    category ← f(state)
    return <lexeme,category>
} else report failure
```

Transition Table (δ)

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Skeleton Scanner Tables

Still O(1)
More Complex REs

What about a more complex language?

• r [0...9] [0...9] allows arbitrary register numbers (e.g., r000 or r999)
• What if we want to limit the register name to r0 through r31?

Write a tighter specification into the RE

• r ( (0|1|2) ([0...9] | ε) | (4|5|6|7|8|9) | (3|30|31) )
• r0|r1|r2|r3| ... |r31|r00|r01|r02| ... |r09

Each of these REs can be converted to a DFA

• The DFA has the same O(1) cost per transition
• The DFA takes one transition per input character
• The DFA uses the same skeleton scanner

The added complexity is in the RE, not in the scanner†
More Complex REs

The DFA for \( r ( (0|1|2) ([0...9] \mid \epsilon) \mid (4|5|6|7|8|9) \mid (3|30|31) ) \)

- Accepts a more constrained set of register names
- Same cost per input character
- More states \( \Rightarrow \) more rows in the transition table \( \Rightarrow \) more memory
More Complex REs

The DFA for \( r \ (0|1|2) ([0...9] \ | \ \epsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) \ )

- Accepts a more constrained set of register names
- Same cost per input character
- More states \( \Rightarrow \) more rows in the transition table \( \Rightarrow \) more memory

Automata Theory Moment
Earlier, we said we would revisit logical complement of an RE or a DFA.
To complement a DFA:
- Make non-final states into final states
- Make final states into non-final states
DFA then accepts any string that the original did not accept \( \Rightarrow \) its complement
More Complex REs

The DFA for \( r \ (0|1|2) ([0...9] \ | \ e) \ | (4|5|6|7|8|9) \ | (3|30|31) \ )

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>( r )</th>
<th>0, 1</th>
<th>2</th>
<th>3</th>
<th>4 \ldots 9</th>
<th>Any Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_2 )</td>
<td>( s_2 )</td>
<td>( s_5 )</td>
<td>( s_4 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_e )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_3, s_4 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_e )</td>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
</tbody>
</table>

Notice that the character classifier has many more divisions than did the earlier one. Still, it should be implementable as a function with \( \mathcal{O}(1) \) cost. (see § 2.5)

This table runs without change in the same skeleton scanner as the first table
- To change the language, just change the table
- Still \( \mathcal{O}(1) \) cost per character